Towards Analyzable Pipeline Analyses

or DESPA: Discrete Event Simulation for Pipeline Analyses

(draft)

Mohamed Abdel Maksoud

November 18, 2011

1 Introduction

In the heart of the WCET analysis lies the pipeline analysis. The pipeline analysis simulates the execution of a task/program based on a timing model of the executing processor. This timing model operates at the granularity of processor cycles. Because of abstractions, this simulation is non-deterministic. To be safe, the analysis has to follow all possible paths. The analysis cannot follow only the locally worse paths because of the presence of timing anomalies. Reineke and Sen presented in [RS09] a formal method to safely discard the non-local-worst-case branches. This is accomplished by precomputing a function for all pairs of possible processor states. Unfortunately, for processor models of reasonable complexity, this precomputation is intractable.

In this work, we propose a formalism to specify processor models in a way that facilitates deriving properties on them that are valid for all program sequences. This formalism is based on the fact that processors consist of different interacting units. In a reasonably sophisticated processor, a unit remains idle in a significant number of processing cycles during program execution, either because it is awaiting input(s) from other unit(s), or because the buffer to which it outputs is full. Any cycle-level simulation (or more formally, activity-based simulation) is bound to represent these idle unit states, resulting in a large state space. Our formalism relies on the mechanism of event-based simulation of the program execution in the pipeline. Using this formalism, the analysis keeps track only of the unit states where operations are accomplished, potentially making the processor model tractably analyzable.

2 Formalism

Pipeline analyses only simulate the timing behavior of the processor, other details are irrelevant. That is, it only matters *how long* some unit takes to perform an operation, not *how* it actually performs the operation. Using this intuition, units in our formalism are generic modules that consume tokens, process them for a certain amount of cycles,

then produce tokens. The interaction between different units is formalized in the next section.

2.1 Producer-Buffer-Consumer State

A Producer-Buffer-Consumer state represents two units, a producer and a consumer, communicating through a bounded buffer. Because we adopt event-based simulation mechanism, producer and consumer can operate at different time points, therefore the state keeps track of the *difference* between the times where each of them started the operation (i.e. *time of start* or *ts*):

Definition 1. *(Producer-Buffer-Consumer state).*
A PBCState $\subseteq \mathbb{Z} \times \mathbb{N}$ *is a tuple (*Δts*, buffer-size) where:*

- Δts *:= consumer.ts - producer.ts, and*
- *buffer-size := number of elements in the buffer between producer and consumer at the time point* min*(producer.ts, consumer.ts)*

∘

During simulation, an $s \in PBCState$ is updated according to the following definition, the term *delay* here refers to the amount of time needed by a unit to complete an operation:

Definition 2. *(PBCState update).*
PBCState-update : PBCState $\times \mathbb{N} \times \mathbb{N} \rightarrow$ *PBCState is defined according to the following function:*

---

function PBCState-update (s, p.delay, c.delay):
let p.ts=0, c.ts=s.Δts, p.ts'=p.ts+p.delay, c.ts'=c.ts+c.delay
find p.ts', c.ts', Δbuffer-size according to the following table:

Case	Conditions	Δbuffer-size
_p.ts_p.ts'_c.ts _c.ts'_	buffer(p.ts) < BUFFER-CAPACITY ? OK : p.ts'=c.ts	+1 : 0
_p.ts_c.ts _p.ts'_c.ts'_	buffer(p.ts) > 0 ? OK : c.ts'=c.ts'+(p.ts'-c.ts)	+0 : 0
_p.ts_c.ts _c.ts'_p.ts'_	buffer(p.ts) > 0 ? OK : c.ts'=c.ts'+(p.ts'-c.ts)	-1 : 0
_c.ts_c.ts'_p.ts _p.ts'_	buffer(c.ts) > 0 ? OK : c.ts'=c.ts'+(p.ts'-c.ts)	-1 : 0
_c.ts_p.ts _c.ts'_p.ts'_	buffer(c.ts) > 0 ? OK : c.ts'=c.ts'+(p.ts'-c.ts)	-1 : 0
_c.ts_p.ts _p.ts'_c.ts'_	buffer(c.ts) > 0 ? OK : c.ts'=c.ts'+(p.ts'-c.ts)	+0 : 0

return PBCState ((c.ts'-p.ts'), s.buffer-size+Δbuffer-size)

∘

In the table in Definition 2, the first case specifies the instance where the producer has to *stall* for a time greater than its delay, because the output buffer is full. The function *producer-stall-time* specifies the amount of time it takes the producer to complete an operation, i.e. to produce a new token:

Definition 3. *(Producer stall time).*
The function producer-stall-time : PBCState $\times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ *is defined as follows:*
producer-stall-time (s, p.delay, c.delay) $:= \begin{cases} s.\Delta ts & \text{fist case holds in Definition 2} \\ p.delay & \text{otherwise} \end{cases}$
○

2.2 Processor Models and States

Definition 4. *(Processor model).*
A processor model $\mathcal{M}$ *is a tuple (ULab, connections, delays), where:*

- *ULab := the set of all unit labels*
- *connections* $\subseteq$ *ULab* $\times$ *ULab* $\times \mathbb{N}$ *:= (u1, u2, buffer-capacity)*
- *delays : ULab* $\rightarrow \mathbb{N}^{+}$

Additionally, we define the following functions:

- *entry-unit (*$\mathcal{M}$*) := the label of the unit where instructions enter the pipeline, restrictions: fan-in=0 and fan-out=1*
- *exit-unit (*$\mathcal{M}$*) := the label of the unit where instructions go after retiring. restrictions: fan-out=0 and unbounded input buffer(s)*
- *unit-level (*$\mathcal{M}$*, ULab) := the distance to exit-unit(*$\mathcal{M}$*), e.g. the unit level of the exit unit is zero.*
- *retire-units(*$\mathcal{M}$*) :=* $\{$*u* $\in$ *ULab: unit-level(*$\mathcal{M}$*, u) = 1*$\}$

○

Definition 5. *(Processor state).*
The set of processor states PState of a processor whose model is $\mathcal{M}$ *is a tuple (pou, udm), where:*

- *pou:* $\mathcal{M}.connections \rightarrow$ *PBCState (pairs of units)*
- *udm: ULab* $\rightarrow \mathbb{N}$ *(unit delay map)*

Additionally, we define the following functions:

- *buffer-occupancy-at-level (*$\mathcal{M}$*, ps* $\in$ *PState, l) :=*
$\sum\{$*PBC-update(ps.pou(c), ps.udm(u1), ps.udm(u2)).buffer-size: c=(u1, u2, CAPACITY)* $\wedge$ *unit-level(*$\mathcal{M}$*, u1)=l*$\}$
- *producer-stall-at-level (*$\mathcal{M}$*, ps* $\in$ *PState, l, p.delay, c.delay) :=*
$\{$*producer-stall-time(ps.pou(c), ps.udm(u1), ps.udm(u2)): c=(u1, u2, CAPACITY)* $\wedge$ *unit-level(*$\mathcal{M}$*, u1)=l*$\}$

○

The update function of a processor state is based on the function PBCState-update defined in the previous section.

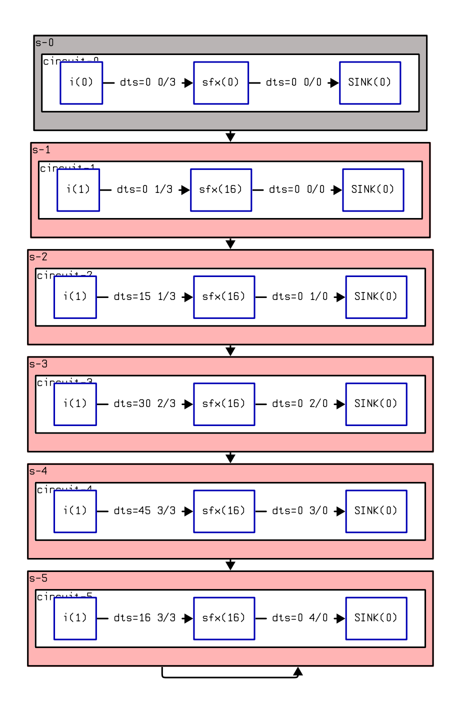


Figure 1: The reachability graph of a simple processor.

2.3 An Example Processor

To illustrate how the simulation proceeds in our formalism, the reachability graph of a simple processor model is depicted in Figure 1. The processor consists of an instruction unit `i` and a simple function unit `sfx`. The instruction unit in our model is very fast (i.e. always instruction cache hit), it fetches a new instruction into the pipeline every cycle. The function unit is not as fast, it takes 16 cycles to process an instruction and pass it on to the `SINK`. The reachability graph is based on a *total order* on processor states. This order is formalized in the following section.

2.4 The Total Order on Processor States

Specifying the processor state in terms of producer-buffer-consumer states allows for defining a total order relation on them. The relation $\prec$ is defined as follows:

Definition 6. *(Ordering relation on processor states).*
For processor states s1, s2 $\in$ PState of a processor whose model is $\mathcal{M}$, s1 $\prec$ s2 (read s1 is less performant than *s2) according to the following function:*
s1 $\prec$ s2 := λ s1, s2 .

for *l in 1 .. nlevels($\mathcal{M}$)* ***do***
  if *l > 1 $\wedge$ buffer-occupancy-at-level($\mathcal{M}$, s1, l) <*
  buffer-occupancy-at-level($\mathcal{M}$, s2, l) ***then***
    true
  end if
  for *t1 in sorted(producer-stall-at-level($\mathcal{M}$, s1, l))*
  t2 in sorted(producer-stall-at-level($\mathcal{M}$, s2, l)) ***do***
    if *t1 > t2* ***then***
      true
    end if
  end for
end for
false

$\circ$

The ordering relation is based on the fact that a state with a full buffer is more performant than one with an empty buffer. The significance of a buffer depends on its distance to the exit unit, e.g. the buffers indicating progress the most are the ones connecting exit unit to the retire units. These *unbounded* buffers are not considered though in the ordering relation since they cannot cause any stall in the pipeline. If two states have the same buffer size valuation at one level, one state is more performant than other if it will produce a new token in the buffer *before* the other, hence is the checking with the function producer-stall-at-level.

3 Discussion and Future Work

In this work we have introduced a novel formalism of modeling pipeline analysis where we adopt an event-based simulation mechanism. The formalism allows for a uniform, systematic development of processor states. The simulation proceeds at the speed of *instruction semantics* rather than the speed of *cycle semantics*. The formalism allows for an intuitive total ordering relation between processor states. The formalism has a potential for reducing the size of the reachability graph, and consequently facilitating the derivation of static properties on the analyses, e.g. the function used to discard timing anomalies in [RS09].

The processor models considered so far are simple, and the method is to be extended for representing real-world processors (e.g. model processors accepting differ-

ent classes of instructions, model units consuming from multiple units, rather than a single one).

References

[RS09] Jan Reineke and Rathijit Sen. Sound and efficient WCET analysis in the presence of timing anomalies. In *Proceedings of 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, June 2009.